Name: Mark Vincent Ty
Student No: 2018-21871

Machine Problem 1 Written Report:

To quickly view the results, open the link:
https://github.com/markytools/EE214MachineProblems/blob/master/machineprob1.ipynb

1. a.) First we created a function called “getN1000Sn” which simulates 1000 repetitions each with 100 coins tossed. Its parameter is how many heads should appear out of the 100. It then returns the total number of trials “totalTrialsEqualN” having the same number of heads. In addition to this, we also return the mean and standard deviation of each 100 tosses.
   We then use this function to acquire the different values of N(S) from 34 to 68 accordingly, where groups of three are summed up accordingly.
   b.) We plot the average heads per group (the middle group) vs the number of trials that have nHeads. For simplicity, we sum up all three groups to have only a single plot instead of plotting three groups separately.
   c.) The results from b are then normalized to turn it into a probability score. The mean expectation and mean standard deviation are printed, along with the probability of heads for each average head per group. We do this another 19 times, calling the “getN1000Sn” function to produce a new 1000 trials.

2. We create a function called “computeRel” that computes the reliability (comp. prob.) of a sequence of components. The function accepts the start and end index since we enter a list of components into it. We can also specify the method (be it Series(S) or Parallel(P)).
   2.1) A function called system_operational assumes a numpy array input having the total 100 components. Components 1-40 and 51-80 are computed in series, while 41-50 and 81-100 are computed in parallel. The total probability is then computed as a sequence of top plus bottom parallel connection. The function outputs the total probability of the system. For convenience, the function also returns the reliability of the two series connections, the two parallel connections, and the top and bottom calculated reliabilities.
   We create two functions called “getSeriesCompToReplace” and “getParallelCompToReplace” that returns the index of the component that needs to be replaced in a series and parallel components, respectively. Simply put, it returns the lowest reliable component in a series, and the highest reliable component in a parallel connection.
   2.2. and 2.3) Given 10 ultra reliable components, we create a function that sets the reliability of 20 selected components. Using the functions “getSeriesCompToReplace” and “getParallelCompToReplace”, we acquire the indices of different components with one another (as according to how the system is made of). We replace these components with a reliability having a failure rate of 1/10 its original failure (q=q/10). The function “replaceWithUltraReliable” initializes the 100 components based on the given problem, then calls the function

replaceCompWithUltra to replace the 20 components needed to be replaced at minimum costs. We print out how the reliability goes down, and then we also compare it with having to replace 20 random components with the ultra reliable ones. We found out that the former produced better overall reliability with the same number of components replaced than the latter method.

3. A python module called pratt_woodruff.py is created. From the paper, we recreate the equation for calculating the result of a binomial pdf replaced by taking the natural log of both sides in the function “getBinomialProbUsingFact”. The function “getBinomialProbUsingGamma” is done by replacing the factorial in “getBinomialProbUsingFact” by the gamma function. Since this produces a math overflow error, we simplify the equation for finer calculation of large numbers (see pratt_woodruff.py for more details). The function “binpmf” calculates the probability of the binomial with parameters n,p, and k. Finally, we plot the probability of the binomial pdf, which almost has a bell shaped curve.

4. We first calculate the constant number c(n), a function of n. It is done by just transposing the pdf of the zipf function equation. After this, we loop through n from 1 to 1000. For each n, we get its corresponding c(n) from the function we made before. We acquire the probabilities of different values of x <= n. We then get the left-tailed probabilities having p(x) <= 0.7, 0.8 or 0.9. This collection of x signify the 70%, 80% or 90% most popular web pages that needs to be cached by the system for each total number of n pages. The endmost page maxK represents the last page number below or equal to the threshold.